Notes from 12/14

```
stack install wai warp
```

```haskell
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Map as M
import qualified Data.Set as S
import Control.Monad.Cont
import Data.IORef
import Data.String.Conversions (cs)
import Control.Exception (bracket_)
import Network.HTTP.Types.Status
import qualified Network.Wai.Handler.Warp as Warp
import Network.Wai
```

Assignment 10 solution

```haskell
data Prob a = Prob { toList :: [(a, Rational)] }
  deriving (Show)

always :: a -> Prob a
always a = Prob [(a, 1)]

oneOf :: [a] -> Prob a
oneOf as = Prob (map g as)
  where g a = (a, 1 / toRational(length as))

optimize :: Ord a => Prob a -> Prob a
optimize (Prob xs) =
  Prob (M.toList (M.fromListWith (+) xs))

instance Functor Prob where
  fmap f (Prob xs) = Prob (map g xs)
    where
      g (a,p) = (f a, p) -- Apply f to value a, preserve probability p

instance Applicative Prob where
  pure = always
  Prob fs <*> Prob xs =
    Prob [(f x, p*q) |(f,p) <- fs, (x,q) <- xs]
```

```haskell
instance Monad Prob where
  Prob as >>= f =
    Prob (concatMap each as)
    where
      each (a,p) = scale p (f a)
      scale p (Prob bs) = map (\(b,q) -> (b, p*q)) bs

data Suit = Hearts | Clubs | Spades | Diamonds
  deriving (Eq, Ord, Show, Enum, Bounded)

allSuits :: [Suit]
allSuits = [minBound..maxBound]

data Rank
  = RankAce
  | Rank2
  | Rank3
  | Rank4
  | Rank5
  | Rank6
  | Rank7
  | Rank8
  | Rank9
  | Rank10
  | RankJack
  | RankQueen
  | RankKing
  deriving (Eq, Ord, Show, Enum, Bounded)

allRanks :: [Rank]
allRanks = [minBound..maxBound]

data Card = Card { cardRank :: Rank
                 , cardSuit :: Suit
                 } deriving (Eq, Ord, Show)

allCards :: [Card]
allCards =
  [Card r s | r <- allRanks, s <- allSuits]

deck :: S.Set Card
deck = S.fromList allCards

type Hand = S.Set Card
type Deck = S.Set Card
```

```haskell
drawThreeCards :: Prob Hand
drawThreeCards = do
  firstCard <- oneOf (S.toList deck) -- 1/52
  let newDeck = S.delete firstCard deck
  secondCard <- oneOf (S.toList newDeck) -- 1/51
  let newNewDeck = S.delete secondCard newDeck
  thirdCard <- oneOf (S.toList newNewDeck)  -- 1/50
  return $ S.fromList [firstCard, secondCard, thirdCard]

isFlush :: Hand -> Bool
isFlush hand =
  S.size (S.map cardSuit hand) == 1

isStraight :: Hand -> Bool
isStraight hand =
  case S.toAscList (S.map cardRank hand) of
    [r1,r2,r3]
      | succ r1 == r2 && succ r2 == r3 -> True
    _ -> False
```

Web services

```haskell
app :: IORef Int -> Application
app counter req respond = bracket_
    (putStrLn "Allocating scarce resource")
    (putStrLn "Cleaning up") $ do
      putStrLn $ show req
      n <- atomicModifyIORef' counter (\i -> (i+1,i))
      respond $ responseLBS status200 [] $
         cs ("Visitor #" ++ show n)

main2 :: IO ()
main2 = do
  ref <- newIORef 0
  Warp.run 8180 $ app ref
```

Continuations

See this blog post.

```haskell
ex1 :: Monad m => m Int
ex1 = do
  a <- return 1
```

```
  b <- return 10
  return $ a+b

ex2 = do
  a <- return 1
  b <- cont (\fred -> fred 10)
  return $ a+b

ex3 = do
  a <- return 1
  b <- cont (\fred -> "escape")
  return $ a+b

ex4 = do
  a <- return 1
  b <- cont (\fred -> fred 10 ++ fred 20)
  return $ a+b

ex5 = do
  a <- return 1
  b <- cont (\fred -> if fred 5 > 100 then 4140 else fred 12)
  return $ a+b

type MyCont r a = (a -> r) -> r

retC :: a -> MyCont r a
retC a k = k a

bindC :: MyCont r a -> (a -> MyCont r b) -> MyCont r b
bindC f g k = f (\a -> g a (\b -> k b))

main :: IO ()
main = return ()
```